Modeling and Reasoning over Distributed Systems using Aspect-Oriented Graph Grammars

Rodrigo Machado
Univ. Federal do Rio Grande do Sul
Porto Alegre, Brazil
rma@inf.ufrgs.br

Reiko Heckel
Univ. of Leicester
Leicester, UK
reiko@mcs.le.ac.uk

Leila Ribeiro
Univ. Federal do Rio Grande do Sul
Porto Alegre, Brazil
leila@inf.ufrgs.br

Aspect-orientation is a relatively new paradigm that introduces abstractions to modularize the implementation of system-wide policies. It is based on a composition operation, called aspect weaving, that implicitly modifies a base system by performing related changes within the system modules. Aspect-oriented graph grammars (AOGG) extend the classic graph grammar formalism by defining aspects as sets of rule-based modifications over a base graph grammar. Despite the advantages of aspect-oriented concepts regarding modularity, the implicit nature of the aspect weaving operation may also introduce issues when reasoning about the system behavior. Since in AOGGs aspect weaving is characterized by means of rule-based rewriting, we can overcome these problems by using known analysis techniques from the graph transformation literature to study aspect composition. In this paper, we present a case study of a distributed client-server system with global policies, modeled as an aspect-oriented graph grammar, and discuss how to use the AGG tool to identify potential conflicts in aspect weaving.

1 Introduction

Aspect-oriented programming [11] is a relatively new paradigm which introduces abstractions to modularize the implementation of system-wide policies. It is based on a composition operation, called aspect weaving, that modifies a base system globally according to structural rules such as, for instance, "register in a global log all modifications in the value of the field x of class C". As characterized in [7], an aspect is a module that *i)* identifies in other modules sets of execution points, which are called *pointcuts*, and *ii)* define transformation rules associated with pointcuts. Those rules are called *advices*. Given a pointcut language expressive enough, the implementation of global policies becomes modular and consistent as the system evolves, i.e. new modules will abide by the global policy in the same way as the current ones.

Those advantages stimulated the adoption of aspect-oriented programming in software development. Several languages now have aspect-oriented extensions, the most popular being the Java superset AspectJ [10]. Moreover, the usage of AOP-related concepts also started to appear in languages for system modeling, such as UML diagrams [9]. However, the wide usage of aspect-oriented concepts also introduces issues in the software development process. The implicit modifications caused by aspect weaving may result in behaviors which are difficult to identify by source code analysis. Also, when the system has more than one aspect they may interfere with each other, resulting in different final systems according to the order they are combined. To deal with those problems, the developer needs proper models to reason consistently about the aspect influence. On the formal side, several aspect-oriented calculi have been proposed to characterize aspect interference over programming languages [16, 8, 5, 4]. On the implementation side, integrated development environments offer support to new views related to aspect weaving [1]. However, outside the scope of source-code level aspects, there are still few models and techniques available to reason about aspect-oriented diagrams.

I. Mackie and A. Martins Moreira (Eds.): Tenth International
Workshop on Rule-Based Programming (RULE 2009)
EPTCS 21, 2010, pp. 39–50, doi:10.4204/EPTCS.21.4

The current proposals for studying aspect weaving over diagrams have a strong connection with graph transformation systems and graph grammars, models where the system state is represented by a graph, and its execution, by the application of graph rewriting rules. Given the natural representation of diagrams as graphs, both aspect-oriented systems and graph transformation systems share some common characteristics: pointcuts resemble matches for graph rules, and advices resemble graph rules themselves. In [13], aspect-oriented graph grammars (AOGGs) were proposed as an extension of the traditional graph grammars, where aspects were modeled as second-order transformations over the original specification. The advantage of this approach is that the same rewriting mechanism is used for both aspect weaving and base system execution, allowing to relate them formally. However, up to now it was not shown how to reason about AOGG models. In this work, we propose the use of AGG [2], an attributed graph grammar specification and analysis tool to reason about AOGG models. Since AGG does not support directly AOGGs, we propose an encoding procedure to convert graph grammar specifications into a single typed graph, with aspects being modeled as sets of graph rewriting rules. Then, we use the analysis functionalities of AGG to study conflicts between advices and aspects in the encoded AOGG system.

The text is organized as follows: in Section 2, we review the graph grammar model, and introduce the base client-server example. In Section 3, we recall aspect-oriented graph grammars and present examples of aspects over the base system. In Section 4 we present the encoding of AOGGs as typed graphs in order to use the AGG tool. We discuss the use of critical pair analysis to study the aspect weaving in the encoded system in Section 5. Final remarks, related work and future steps are discussed in Section 6.

2 Graph Grammars

A typed graph grammar is a visual model where the system state is represented by a graph and the system behavior is described by means of rule-based graph rewriting. Formally, a typed graph grammar is a tuple $\langle T, G_0, P, \pi \rangle$, where *i)* T is a type graph defining the allowed kinds of nodes and edges within the specification; *ii)* the T-typed graph G_0 is the initial state of the system; *iii)* P is a set of rule names, and *iv)* the function $\pi : P \to Rules(T)$ maps every rule name to is respective T-typed graph transformation rule. In this work we follow the double-pushout (DPO) approach to graph transformation [6], where a graph transformation rule is represented as a monic span (a pair of injective morphisms with the same source) $L \overset{l}{\hookleftarrow} K \overset{r}{\rightarrowtail} R$ in the category T-**Graph** (T-typed graphs and type-preserving graph homomorphisms). The left-hand side (LHS) graph L represents the pattern to be found in order to apply the rule. The right-hand side (RHS) graph R defines new graph elements to be added by the rule. The interface graph K and graph inclusions l and r identify elements in L and R. Elements in L but not in $l(K)$ are said to be *deleted*, elements in R but not in $r(K)$ are said to be *created*, and elements in K are *preserved* or *read*.

Example 1 (Graph grammar)**.** *Figure 1 depicts a distributed client-server system modeled as a graph grammar. The type graph* `T` *defines four kinds of nodes (*`Client`*,* `Server`*,* `Data` *and* `Message`*) and four kinds of edges. The initial graph G_0 defines the initial state of the system: two clients with messages to be sent and two data servers. Clients may retrieve values from servers or update stored data by means of message exchange. The system rules model this interaction as follows: a* `GET` *message is sent to the server by* `SendGET`*, then the data is retrieved by* `ExecuteGET`*, and it is returned to the client by* `ReceiveGET`*. The rules* `SendSET`*,* `ExecuteSET` *and* `ReceiveSET` *work in a similar way for messages that update data elements. Regarding the notation, rules are presented only by their left-hand side and right-hand side graphs. Common elements in K are simply marked to be the same in L and R using*

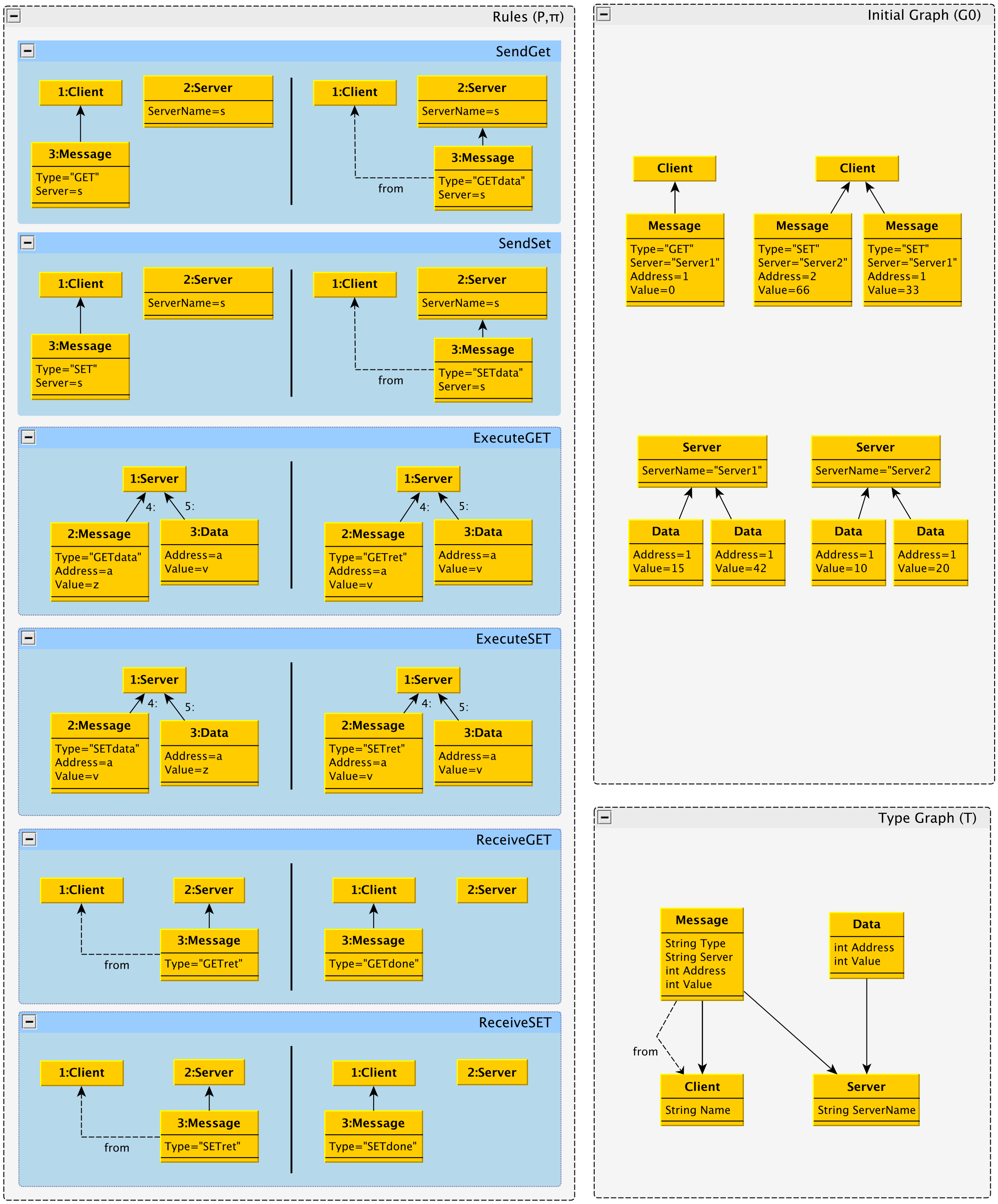


Figure 1: Client-server system as a graph grammar.

numbered prefixes (1:, 2:, ...)*. It is also important to notice that this example uses attributed graph nodes, which is an extension of the basic typed graph grammar formalism.*

In order to apply a graph transformation rule $L \leftarrow K \rightarrow R$ over a graph G it is required that there is some *match* in G for elements of L. Matches are defined as graph homomorphism $m : L \rightarrow G$. Then, a rule application (or direct derivation) from graph G to graph H using rule r and match m (denoted $G \overset{r,m}{\Longrightarrow} H$) is defined as the following diagram in the category T**-Graph**, where both squares are pushouts.

$$\begin{tikzcd}
L \arrow[d, tail, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, tail, "l"'] \arrow[r, tail, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*"] \\
G & D \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

In the DPO approach, rule application relies on finding $G \overset{l^*}{\leftarrow} D \overset{k}{\leftarrow} K$ such that the square to the left is a pushout. This may not always be possible depending on the rule and match. For instance, when the rule tries to delete a node that has incident edges not in $m(L)$, or when the match identifies preserved elements with deleted ones.

The sequential behavior of a graph grammar is given by the successive application of graph rules starting from the initial graph. Usually, both rules and matches are nondeterministically chosen at each step. Operationally, this works as follows:

1. Set G_0 as the current graph.
2. Find in the current graph all possible matches for rules in the set P of rule names.
3. Verify application conditions for matches, defining the set of valid pairs (rule,match).
4. If there is no valid pair at all, then STOP. Otherwise, non-deterministically choose a pair (rule,match) to be applied.
5. Delete from G all matched elements in $L \setminus K$. This will generate a graph D.
6. Create in D all elements in $R \setminus K$. This will generate a graph H.
7. Set H as the current graph. Return to step 2.

Graph grammars are particularly useful to represent distributed systems since graph rules represent local transformations. This means that two or more rules may be applied in parallel if they do not conflict with each other. Moreover, the rules are usually intuitive and the control flow is data-driven, being guided by the graph topology.

The AGG tool [2] allows to create, run and analyze graph grammars. Moreover, it supports several extensions to the basic typed graph grammar language, such as attributed graph elements, control flow mechanisms (rule layers, priorities), definition of positive and negative application conditions for rules, among others. The AGG graph transformation engine follows the single-pushout (SPO) approach [12], but it may also be configured to consider the conditions that characterize DPO transformations.

3 Aspect-Oriented Graph Grammars

Aspect-Oriented Graph Grammars (AOGGs) [13] are graph grammars with aspects, i.e. modular descriptions of system-wide policies. Formally, an AOGG is a pair $\langle \mathscr{G}, \Delta \rangle$, where $\mathscr{G} = \langle T, G_0, P, \pi \rangle$ is a

base graph grammar, and $\Delta = [A_1, A_2, \ldots, A_n]$ is a sequence of graph aspects over $\mathcal{G}$. A graph aspect represents a set of modifications over the base graph grammar, and it is defined by a triple $\langle D, t, g \rangle$, where D is a set of graph advices, $t : T \hookrightarrow T'$ is an extension of the original type graph and $g : G_0 \hookrightarrow G'$ is an extension of the original initial graph.

Graph advices are rules that modify graph transformation rules. Following the intuition provided by the DPO approach, graph advices are defined as monic spans $p \leftarrowtail i \rightarrowtail e$. The difference is that p, i, e are themselves graph rules, and l, r are rule morphisms. Rule morphisms are defined as triples of morphisms connecting the graph components of two rules, respecting the commutativity of the inner squares. This way, a graph advice correspond to the following commutative diagram in T-**Graph**.

$$\begin{tikzcd}
 & & R_p & & & R_i \arrow[lll, tail] \arrow[rrr, tail] & & & R_e \\
 & K_p \arrow[ur, tail] \arrow[dl, tail] & & & K_i \arrow[lll, tail] \arrow[rrr, tail] \arrow[ur, tail] \arrow[dl, tail] & & & K_e \arrow[ur, tail] \arrow[dl, tail] & \\
L_p & & & L_i \arrow[lll, tail] \arrow[rrr, tail] & & & L_e & &
\end{tikzcd}$$

To emphasize the distinction between advices and base system rules, the components of a graph advice receive specific names: *pointcut* p, *advice interface* i and *effect* e.

Example 2 (Log Aspect)**.** *Suppose we want to implement a logging mechanism over the client-server system such that every operation leaves an execution trace over a global object (the system logger). To implement this functionality, we should extend the type graph to introduce the logger type, initialize it in the initial graph and modify all rules to register their execution in this global object. Using AOGG, all these modifications can be enclosed within one single aspect, as shown in Figure 2. This aspect has only one advice, which has an empty pointcut. The advice adds a* `Logger` *object to both the LHS and RHS of the rule such that the occurrence on the RHS has the rule name appended to the* `story` *string. Notice that this value update depends on the existence of a reflective operation* `rulename()` *which provides the name of the rule being executed. The empty pointcut matches all possible rules of the specification, thus after aspect weaving all of them will read and update the global log object.*

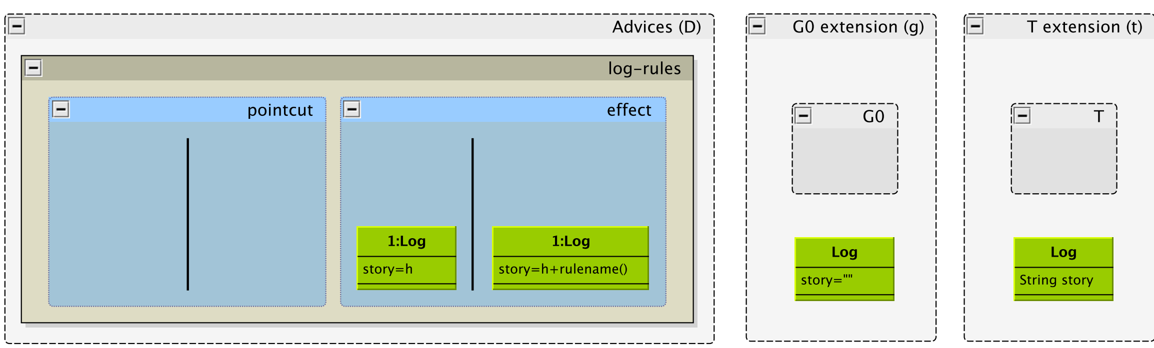


Figure 2: Log aspect

Example 3 (Security Aspect)**.** *Another kind of system-wide modification is the introduction of access control for servers. A simple implementation is to define two kinds of permissions for clients:* `Read` *and* `Write`*. Those permissions, represented as endoarcs in client nodes, affect message exchanges. A "SET"*

message may only be sent by a user with write privileges, while a `"GET"` *message, only by a user with read privileges. The aspect depicted in Figure 3 implements this policy over the base graph grammar of Example 1. The act of sending a message of a given kind (*`SET` *or* `GET`*) defines the pattern used in the* pointcuts *of both advices. The effect is to augment the rules such that they require the respective user permissions to be read in order to execute. In both advices the interface i is the same as the pointcut p.*

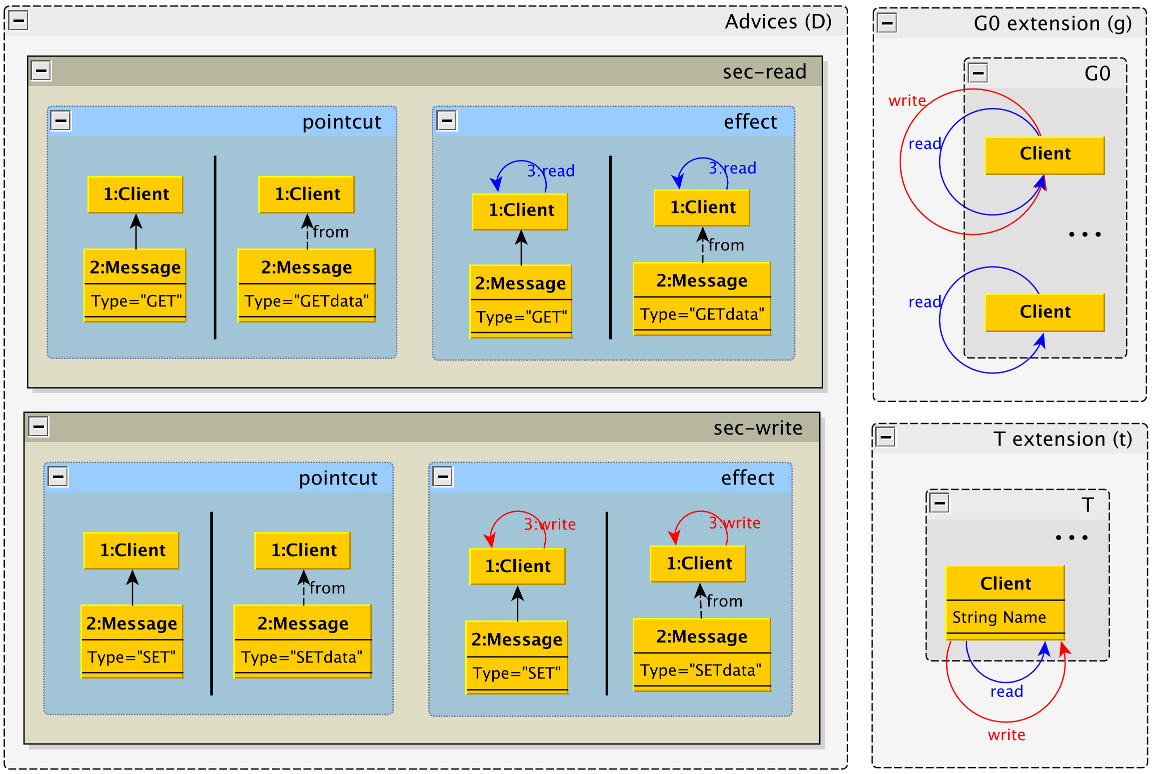


Figure 3: Security aspect

As with graph rewriting, rule rewriting (advice application) is also defined as a double-pushout diagram. The difference is that it uses the category T-**MSpan** of T-typed rules and rule morphisms [13]. Given a graph grammar $\mathscr{G} = \langle T, G_0, P, \pi \rangle$ and a graph aspect $\langle D, t, g \rangle$ over $\mathscr{G}$, with $t : T \rightarrow T'$ and $g : G_0 \rightarrow G_0'$, the result of weaving A and $\mathscr{G}$ is the graph grammar $\mathscr{G}' = \langle T', G_0', P', \pi' \rangle$. The rule set (P', π') is calculated based on (P, π) as the *least* set of rules such that

1. If a rule in (P, π) is not matched by any advice in A, them the rule is kept unchanged in (P', π').
2. If a rule is matched by at least one advice in A, all the results of one-step rewritings (considering all possible advices and matchings) become elements of (P', π'). In this case, the original rule does not appear in (P', π').

This definition for aspect weaving makes advice rewriting non-reentrant, i.e. a given rule may not be modified more than once per advice and match. This assures termination for the weaving process, even for advices that do not delete elements from rules. Finally, the semantics of an AOGG $\langle \mathscr{G}, \Delta \rangle$ is given by its respective weaved graph grammar $\mathscr{G}_W^{\Delta}$, defined as the result of weaving all aspects of Δ over $\mathscr{G}$ in their respective order.

4 Encoding AOGGs in AGG

The use of graph rewriting as an aspect weaving mechanism opens the possibility of using known techniques from the graph transformation area to reason about aspects. Those techniques allow the system modeler to identify potentially harmful behaviors of the final weaved system, such as unintended aspect interactions, at early stages of the system development. Furthermore, existing tools such as AGG can be used to analyze aspect-oriented models.

Our approach consists in encoding the whole structure of a graph grammar specification as a single typed graph. For this, a representation of monic spans using graph elements is needed to encode individual rules. Moreover, one needs to encode both initial graph and rule set while keeping their as distinguished elements of the specification. For this, we employ the traditional graph typing mechanism: the type graph for the encoding of $\mathscr{G} = \langle T, G_0, P, \pi \rangle$ has two disjoint components. The first is the original type graph T, used to type the initial graph G_0. The second is $R(T)$, used to type the encoded representation of the rule set. We start by defining the operation R that calculates the second component based on the original type graph T.

Definition 1 (Type graph rule encoding)**.** *The operation R :* **Graph** $\rightarrow$ **Graph** *is defined as the composition $R_3 \circ R_2 \circ R_1$, where:*

- *R_1 encodes edges as nodes. The edges of the resulting graph follow the source and target functions of the input graph.*
- *R_2 creates three copies of the input graph (to represent L, K and R). Each node in K is connected to its copy in L and R by means of new edges.*
- *R_3 creates a new node in the graph together with edges connecting all the original nodes to this node. This node represents the identity of the rule.*

Example 4 (Type graph rule encoding)**.** *Figure 4 shows how to obtain $R(T)$ from a simple type graph T using the successive transformations R_1, R_2 and R_3.*

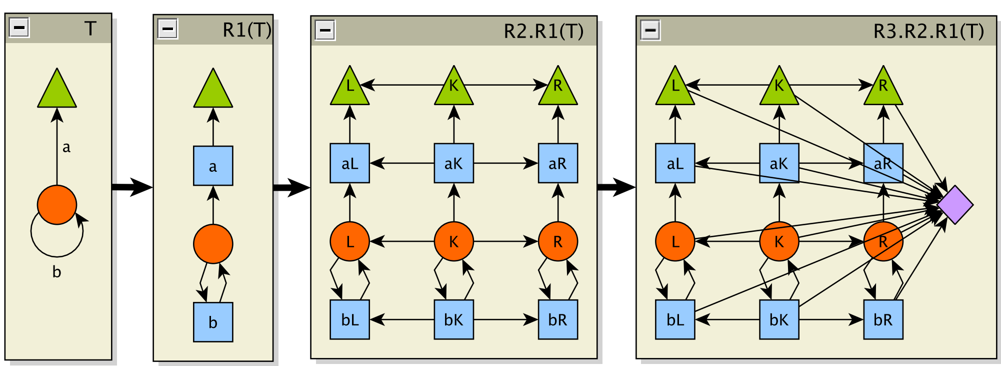


Figure 4: Example of type graph rule encoding.

Now we define the operation S, which encodes T-typed rules as $R(T)$-typed graphs.

Definition 2 (Graph rule encoding)**.** *Given a T-typed graph rule $r : L \xleftarrow{l} K \xrightarrow{r} R$, the graph rule encoding $S : Rules(T) \rightarrow R(T)$-***Graph** *is defined as the composition $S_3 \circ S_2 \circ S_1$, where*

- *S_1 applies R_1 to L,K,R, and types them, respectively, over the left, central and right embeddings of $R_1(T)$ in $R(T)$.*
- *S_2 adds concrete edges to the input graph following the homomorphisms l and r.*
- *S_3 creates a new node typed over the node created by transformation R_3, together with new edges connecting all nodes of the input graph to it.*

Definition 3 (Graph rule set encoding)**.** *Given a graph grammar $\mathscr{G} = \langle T, G_0, P, \pi\rangle$, the $R(T)$-typed graph $\Upsilon(\mathscr{G})$ is defined as the gluing $[S(\pi(r_1)), S(\pi(r_2)), \ldots, S(\pi(r_n))]$ where $r_{1..n} \in P$.*

The graph $\Upsilon(\mathscr{G})$ is the union of all encodings of rules $S(\pi(r_x))$ typed over the single type graph $R(T)$.

Example 5 (Graph rule set encoding)**.** *Figure 5 depicts a small graph grammar $\mathscr{G}$ (on the left) and its respective $R(T)$-typed encoding $\Upsilon(\mathscr{G})$ (on the right).*

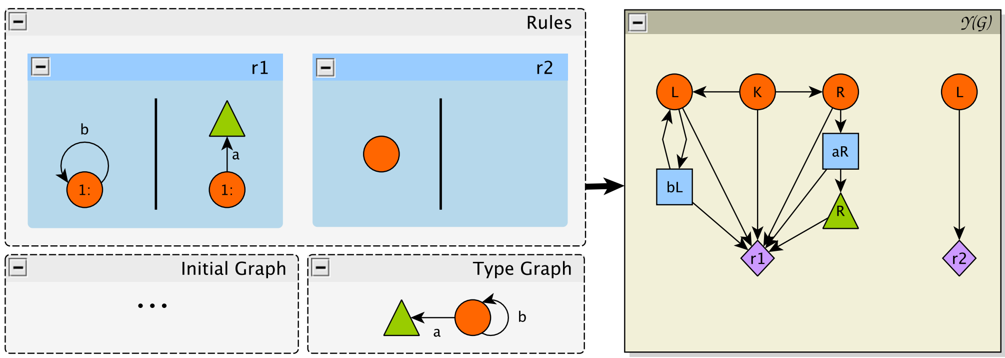


Figure 5: Example of graph rule set encoding.

Definition 4 (Encoded graph grammar)**.** *Given a graph grammar $\mathscr{G} = \langle T, G_0, P, \pi\rangle$, its graph encoding is the $(T + R(T))$-typed graph $|\mathscr{G}| = G_0 + \Upsilon(\mathscr{G})$, where "$+$" stands for the disjoint union of typed graphs.*

Definition 5 (Encoded advice)**.** *Let $A = \langle D, t, g\rangle$ be and aspect over $\mathscr{G}$, such that $t : T \to T'$. Then, for every advice $a : p \leftarrow i \to e \in D$, its respective encoding $|a|$ is the $R(T')$-typed span $S(p) \leftarrow S(i) \to S(e)$.*

Definition 6 (Encoded aspect-oriented graph grammar)**.** *Let $\mathscr{D} = (\mathscr{G}, \Delta)$ be an AOGG, such that $\mathscr{G} = \langle T, G_0, P, \pi\rangle$ and $\Delta = [A_1, A_2, \ldots, A_n]$. Let $\mathscr{G}^\Delta_W = \langle T^\Delta_W, G^\Delta_W, P^\Delta_W, \pi^\Delta_W\rangle$ be the weaved graph grammar obtained from weaving Δ over $\mathscr{G}$. Then, the encoded aspect-oriented graph grammar $|\mathscr{D}|$ is the graph grammar $\langle T^\Delta_W + R(T^\Delta_W), |\mathscr{G}'|, |P_\Delta|, |\pi_\Delta|\rangle$ where $\mathscr{G}' = \langle T^\Delta_W, G^\Delta_W, P, \pi\rangle$ is the type and initial graph extension of $\mathscr{G}$ and the rule set $(|P_\Delta|, |\pi_\Delta|)$ correspond to the gluing of all encoded advices $|a| \in D_i$ of all aspects $A_i = \langle D_i, t_i, g_i\rangle$, $1 \leq i \leq n$.*

In an AOGG $\mathscr{D}$, each aspect extend the type graph and initial graph of the original system. In the encoded graph $|\mathscr{D}|$, all these extensions must be present in order to accommodate all encoded advices (of all aspects) in the same rule set.

Example 6 (Encoded AOGG)**.** *Figure 6 shows the encoding of $(\mathscr{G}, [A_1, A_2])$, where $\mathscr{G}$ is the base system of Figure 1, A_1 is the log aspect (Figure 2) and A_2 is the security aspect (Figure 3). For the sake of readability, some elements are omitted: attribute types in the type graph and attribute values in the initial graph and base system rules. However, attribute values are shown in the encoded advices. In all encoded advices the interface graph K and the left-hand side graph L are the same.*

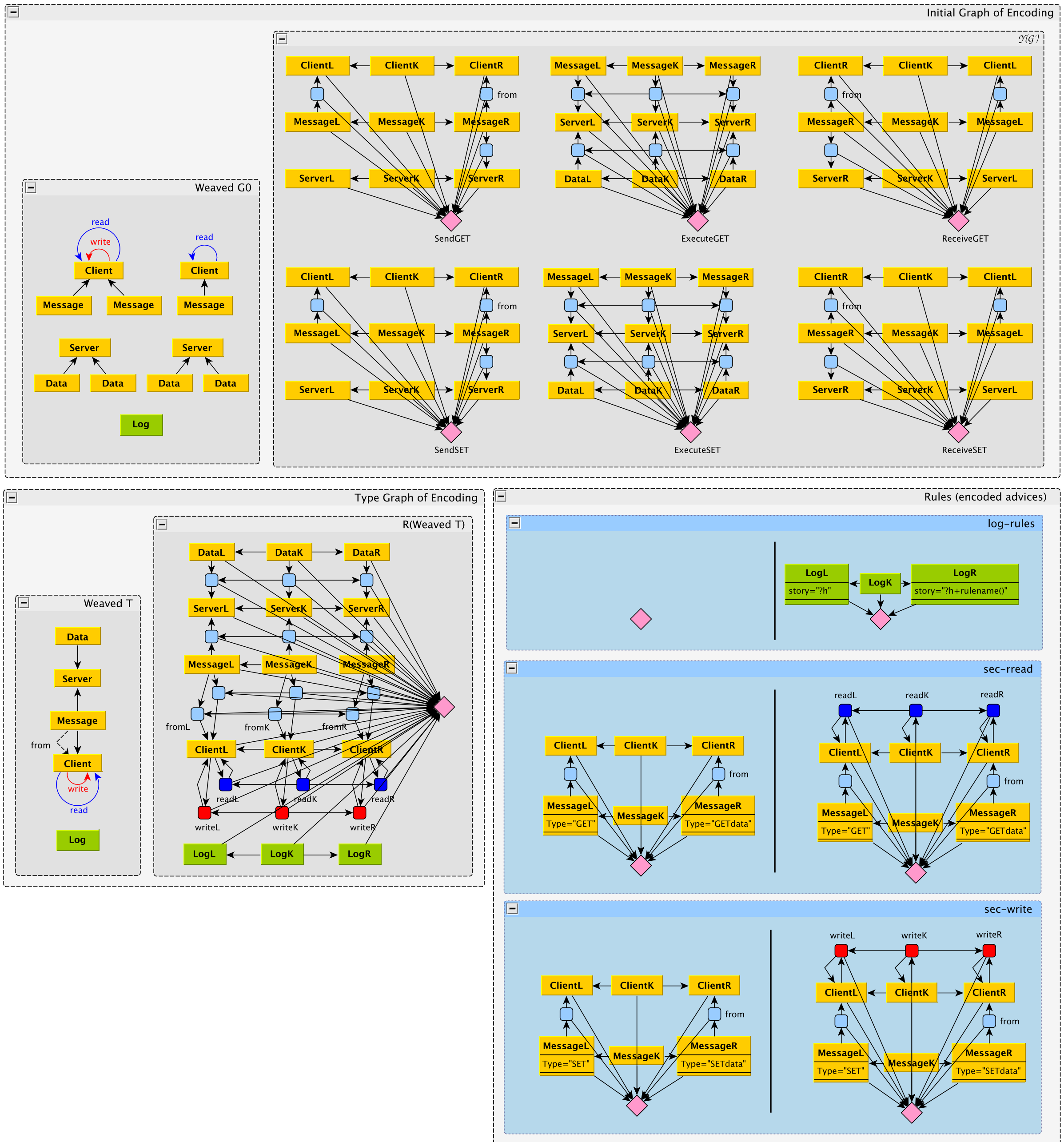


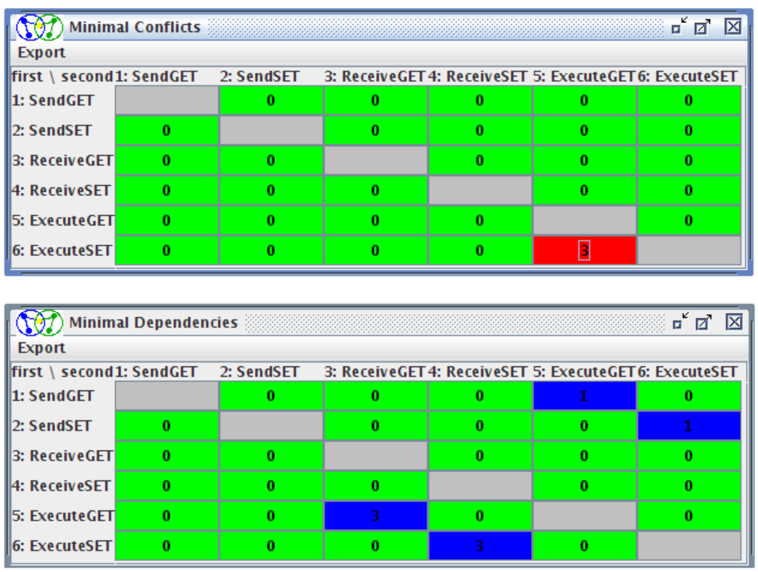
Figure 6: Encoded client-server system with log and security aspects.

5 Analyzing AOGG specifications

From the point of view of aspect weaving, it is relevant to be able to predict when two different advices will potentially affect the same part of the specification. If this occurs, a different aspect ordering may yield a different weaved system. Given that we encode an AOGG specification as a graph grammar, we are able to apply techniques from graph rewriting theory to test the existence of conflicts between aspects.

Critical pair analysis (CPA) is a static analysis technique that is used to detect both conflicts and dependencies between graph rewriting rules. Given a graph G and two direct derivations $G \overset{r_1,m_1}{\Longrightarrow} G'$, $G \overset{r_2,m_2}{\Longrightarrow} G''$, we say that the derivations are in conflict if one of them deletes some graph element that is preserved or deleted by the other one. Thus, their execution in parallel is not possible. Given two subsequent direct derivations $G \overset{r_1,m_1}{\Longrightarrow} G' \overset{r_2,m_2}{\Longrightarrow} G''$, we say that there is a dependency between (r_1,m_1) and (r_2,m_2) if the first creates a graph element that is preserved or deleted by the later. In this case, it is not possible to swap their order of execution. Critical pair analysis is used to test potential conflicts and dependencies within a graph grammar and consists of the following general steps:

1. The set of all pairs of rules $(r_1,r_2) \in P \times P$ is generated.
2. For each pair (r_1,r_2), all possible overlaps between a graph of r_1 and a graph of r_2 are calculated. In the case of conflict analysis, both *LHS*s are considered in this calculation. In the case of dependencies analysis, the *RHS* of r_1 and the *LHS* of r_2 are considered.
3. All overlaps are tested for conflicts or dependencies.
4. The number of all conflicting (or dependent) overlaps for each pair of rules is shown in a table.


Minimal Conflicts

Export

first \ second	1: SendGET	2: SendSET	3: ReceiveGET	4: ReceiveSET	5: ExecuteGET	6: ExecuteSET
1: SendGET		0	0	0	0	0
2: SendSET	0		0	0	0	0
3: ReceiveGET	0	0		0	0	0
4: ReceiveSET	0	0	0		0	0
5: ExecuteGET	0	0	0	0		0
6: ExecuteSET	0	0	0	0	3	

Minimal Dependencies

Export

first \ second	1: SendGET	2: SendSET	3: ReceiveGET	4: ReceiveSET	5: ExecuteGET	6: ExecuteSET
1: SendGET		0	0	0	1	0
2: SendSET	0		0	0	0	1
3: ReceiveGET	0	0		0	0	0
4: ReceiveSET	0	0	0		0	0
5: ExecuteGET	0	0	3	0		0
6: ExecuteSET	0	0	0	3	0	

Figure 7: Critical pair analysis of the base system in AGG.

Aspect-oriented graph grammars are two-layer specifications where the base graph grammar describe the original system behavior, while aspects describe the implementation of global policies as model transformation. By using the encoding of Section 4, we can use AGG to study characteristics of aspect weaving in a similar way as the original system execution. Figure 7 shows the results

of critical pair analysis for the base graph grammar. In this level, conflicts arise only in the pair (`ExecuteSet`,`ExecuteGET`) when the first rule updates a value which is read by the second one. Dependencies reflect `Send-Execute-Receive` sequences of derivations, induced by the value of the `Type` attribute in message nodes. In the aspect level we analyze the possibility of conflicts in the aspect weaving process, since the rules are actually encoded advices. Figure 8 shows the results of CPA for the encoded model of Figure 6. In this particular system, where the advices do not delete elements from rules, the rules are obviously conflict-free. Lack of conflicts in the aspect level means that final weaved system does not change even if we change the order aspects are weaved to the base system. If there was at least one conflicting pair of encoded advices (a_1, a_2) where a_1 and a_2 are not in the same aspect, then the aspects could potentially interact and aspect weaving would not be necessarily a confluent operation.

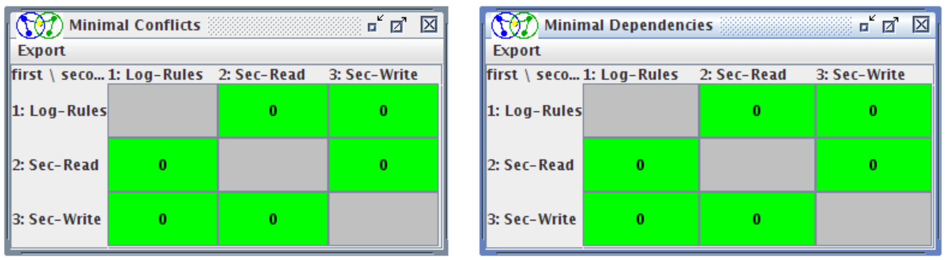


Figure 8: Critical pair analysis of the aspect-oriented encoded system in AGG.

6 Final remarks

This work proposed the use of AOGG to model distributed system with aspects, and also the use of the AGG tool to study their behavior. We introduced the AOGG model, described an encoding mechanism to describe the base system as a single typed graph, and discussed the use of critical pair analysis to reason about conflicts in the aspect weaving operation.

The idea of using graph rewriting to define aspect weaving is not entirely new. In [14, 17], aspect weaving is also defined by means of graph rewriting, and the AGG tool is used to analyze the specification using critical pair analysis. In those approaches, however, UML diagrams are used as base system model. Following a different direction, the work by Aksit et al. [3] encodes a complete aspect-oriented calculus into the graph transformation tool Groove [15]. This allows to study the interference between aspects directly in the system unfolding, by employing the space-state generation feature of Groove.

As future work, we intend to explore the properties of the proposed encoding. This is required in order to relate the behavior of the weaved system to the behavior of the original base system together with aspect specification. Furthermore, the usage of graph grammars as base systems may provide insights about how rule-base modification of rules affect the respective rule derivations, since both are representable as diagrams within the same categorical framework.

Acknowledgements: The authors would like to thank the anonymous referees for their helpful comments and suggestions. This work was partially supported by CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico.

References

[1] *Eclipse AJDT: AspectJ Development Tools*. `http://www.eclipse.org/ajdt/`.

[2] *The AGG Homepage*. `http://user.cs.tu-berlin.de/ gragra/agg/`.

[3] Mehmet Aksit, Arend Rensink & Tom Staijen (2009): *A graph-transformation-based simulation approach for analysing aspect interference on shared join points*. In: *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*. ACM, New York, NY, USA, pp. 39–50.

[4] Curtis Clifton & Gary T. Leavens (2006): *MiniMAO$_1$: an imperative core language for studying aspect-oriented reasonings*. *Sci. Comput. Program.* 63(3), pp. 321–374. Available at `http://dx.doi.org/10.1016/j.scico.2006.02.009`.

[5] Simplice Djoko Djoko, Rémi Douence, Pascal Fradet & Didier Le Botlan (2006): *CASB: Common Aspect Semantics Base,*. Technical Report, Research Report, Network of Excellence in AOSD (AOSD-Europe, August 2006, no D54).

[6] H. Ehrig, M. Pfender & H. J. Schneider (1973): *Graph-grammars: An algebraic approach*. In: *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*. pp. 167–180.

[7] Robert E. Filman & Daniel P. Friedman (2000): *Aspect-Oriented Programming is Quantification and Obliviousness*. Technical Report.

[8] Radha Jagadeesan, Alan Jeffrey & James Riely (2006): *Typed parametric polymorphism for aspects*. *Sci. Comput. Program.* 63(3), pp. 267–296.

[9] José Uetanabara Júnior, Valter Vieira Camargo & Christina Von Flach Chavez (2009): *UML-AOF: a profile for modeling aspect-oriented frameworks*. In: *AOM '09: Proceedings of the 13th workshop on Aspect-oriented modeling*. ACM, New York, NY, USA, pp. 1–6.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William G. Griswold (2001): *An Overview of AspectJ*. In: *ECOOP*. pp. 327–353. Available at `http://link.springer.de/link/service/series/0558/bibs/2072/20720327.htm`.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier & John Irwin (1997): *Aspect-Oriented Programming*. In: *ECOOP*. pp. 220–242. Available at `http://link.springer.de/link/service/series/0558/bibs/1241/12410220.htm`.

[12] Michael Löwe (1993): *Algebraic approach to single-pushout graph transformation*. *Theor. Comput. Sci.* 109(1-2), pp. 181–224.

[13] Rodrigo Machado, Luciana Foss & Leila Ribeiro (2009): *Aspects for Graph Grammars*. In: *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques*.

[14] Katharina Mehner, Mattia Monga & Gabriele Taentzer (2006): *Interaction Analysis in Aspect-Oriented Models*. *Requirements Engineering, 14th IEEE International Conference* , pp. 69–78.

[15] Arend Rensink (2004): *The GROOVE simulator: A tool for state space generation*. In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE), volume 3062 of Lecture Notes in Computer Science*. Springer, pp. 479–485.

[16] David Walker, Steve Zdancewic & Jay Ligatti (2003): *A Theory of Aspects*. In: *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. ACM Press, New York, NY, USA, pp. 127–139.

[17] Jon Whittle & Praveen K. Jayaraman (2007): *MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation*. In: Holger Giese, editor: *MoDELS Workshops*, *Lecture Notes in Computer Science* 5002. Springer, pp. 16–27. Available at `http://dblp.uni-trier.de/db/conf/models/models2007w.html#WhittleJ07`.